Algorithmen und Datenstrukturen

January 8, 2023

Inhaltsverzeichnis

1 Mathematische Grundlagen
- 1.1 Reihen
- 1.2 Potenzen und Logarithmen
- 1.3 Notationskonventionen
- 1.4 Grundbegriffe der Graphentheorie

2 Rekursive Algorithmen
- 2.1 Prinzip der Rekursion
- 2.2 Korrektheit rekursiver Algorithmen
- 2.3 Rekursive Berechnung der Potenzmenge
- 2.4 Algorithmenprinzip "Backtracking"

3 Analyse von Algorithmen
- 3.1 Korrektheit
- 3.2 Komplexität von Algorithmen
- 3.3 Komplexitätsanalyse wohlstrukturierter Algorithmen

4 Sortierverfahren
- 4.1 Vergleichsbasierte Sortierverfahren
- 4.2 Heapsort
- 4.3 Rekursives Sortieren nach "Teile und Herrsche"
- 4.4 Quicksort
- 4.5 Mergesort
- 4.6 Eigenschaften der Sortierverfahren
- 4.7 Nicht-vergleichsbasierte Sortierverfahren

5 Einfache Datenstrukturen
- 5.1 Abstrakte und konkrete Datenstrukturen
- 5.2 Abstrakter Datentyp Stack
- 5.3 Konkreter Datentyp ArrayStack
- 5.4 Amortisierte Laufzeitanalyse
- 5.5 Abstrakter Datentyp Warteschlange
- 5.6 Konkreter Datentyp Ringpuffer
- 5.7 Abstrakter Datentyp Prioritätswarteschlange
- 5.8 Konkreter Datentyp verkettete Liste
- 5.9 Abstrakter Datentyp Double-Ended-Queue (Deque)
- 5.10 Zusammenfassung: Anwendung verketter Listen

6 Suchbäume
- 6.1 Abstrakte Datentypen Map und Set
- 6.2 Binäre Bäume
- 6.3 Binäre Suchbäume
- 6.4 Grundprizip balancierter Suchbäume
- 6.5 AVL-Bäume
- 6.6 B-Bäume
- 6.7 Rot-Schwarz Bäume

1 Mathematische Grundlagen

1.1 Reihen

Arithmetische Reihe

- **Allgemeine arithmetische Reihe**: $a_0+(a_0+d)+(a_0+2d)+ \ldots +(a_0+n \cdot d)$

  $$\sum_{i=0}^{n}(a_0+ i \cdot d)=(n+1)(a_0+ d\frac{n}{2})$$

  Beispiel: Summe der ungeraden Zahlen von 1 bis 99, d.h.1 + 3 + 5 + ... + 99:
  $a_0 = 1$
  $d = 2$
  $n = 49$
  Ergebnis: 50 * (1 + 2*49/2) = 2500

- **Gaußsche Summenformel**: 1 + 2 + 3 + ... + *n*, also Summe der natürlichen Zahlen von 1 bis *n*. Dies ist der Spezialfall mit $a_0 = 0$; $d = 1$.

  $$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

  Beispiel: Summe der Zahlen von 1 bis 50:
  50 * 51 / 2 = 1275

wichtig

1.2 Potenzen und Logarithmen

Der Logarithmus ist die Inverse der Potenzfunktion. $\log_a(x) = y \iff a^y = x$
spezielle Logarithmen:
$ld(x) = log_2(x)$, $lg(x) = log_10(x)$, $ln(x) = log_e(x)$

1.3 Notationskonventionen

$\lceil x \rceil$ zur nächsten ganzen Zahl aufrunden
$\lfloor x \rfloor$ zur nächsten ganzen Zahl abrunden
$[a..b] = x | a \leq x \wedge x \leq b$ mit Intervallgrenzen
$]a..b[= x | a < x \wedge x < b$ ohne Intervallgrenzen
$arr[i..k]$ Teilfolge der Elemente von $arr[i]$ bis $arr[k]$

1.4 Grundbegriffe der Graphentheorie

Graphen bestehen aus einer Menge von Knoten und Kanten, die diese verbinden.
Ein Graph ist gerichtet, wenn die Kanten eine Richtung haben.
Für einen Knoten v eines gerichteten Graphen $G = (V, E)$ ist der Eingangsgrad $indeg(v)$ die Anzahl der Kanten, die in v enden, und der Ausgangsgrad $outdeg(v)$ die Anzahl der Kanten, die von v ausgehen.
Ein Zyklus ist ein Weg der bei einem Knoten startet und endet.
Ein gerichteter Graph ist zusammenhängend, wenn es einen Weg zwischen jedem Knotenpaar gibt.
Ein Baum hat einen Knoten als Wurzel, jeder Knoten hat genau einen Vorgänger und ist zusammenhängend.
Ein Knoten ohne Kinder heißt Blatt. Ein leerer Baum hat die Höhe 0. Ein Binärbaum ist ein Baum, dessen Knoten maximal zwei Kinder haben.
Traversierungen: Preorder (WLR), Inorder (LWR), Postorder (LRW)

2 Rekursive Algorithmen

2.1 Prinzip der Rekursion

Ein rekursiver Algorithmus besteht aus einem Basisfall und einem rekursiven Aufruf.
Der rekursive Aufruf muss immer kleiner werden, damit die Rekursion endet.
Die Rekursion kann durch eine Schleife ersetzt werden.

```
  public static double sum_v2(double[] arr) {
  return sum_v2(arr, 0, arr.length-1);
  }
/** Berechnet Summe der Werte von arr[firstIndex..lastIndex] */
  private static double sum_v2(double[] arr, int firstIndex,int lastIndex) {
  if (firstIndex == lastIndex) {
    // zu summierender Bereich besteht nur aus einem Element
    return arr[firstIndex];
  }
  else {
    int mid = (firstIndex + lastIndex) / 2;
    return sum_v2(arr, firstIndex, mid) + sum_v2(arr, mid+1, lastIndex);
  }}
```

2.2 Korrektheit rekursiver Algorithmen

Ein Beweisverfahren ist die Berechnungsinduktion.

Beweis mittels Berechnungsinduktion. Eigenschaft, die nachgewiesen werden soll:

$P((x,n),y) :\Leftrightarrow x^n = y$

d.h. wir wollen zeigen, dass $h(x,n) = x^n$ gilt.

- **Induktionsbasis** (Argumente führen zu keinem rekursiven Aufruf)
  Fall n = 0:
  Ergebnis: $h(x, n) = 1 = x^0$, d.h. Eigenschaft erfüllt
- **Induktionsschritte** (Argumente führen zu rekursiven Aufrufen):
  Wir können als Induktionshypothese für die rekursiven Aufrufe verwenden, dass $h(z_i,k_i) = z_i^{ki}$ gilt.

  Fall n > 0, n gerade:
  Ergebnis laut Programmcode: h(x,n) = y*y, wobei y = h(x, n/2)

  Wir verwenden die Induktionshypothese: es gilt $h(x, n/2) = x^{n/2}$

  somit $h(x,n) = y*y = x^{n/2} * x^{n/2} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt.

  Fall n > 0, n ungerade:
  Ergebnis laut Programmcode: h(x,n) = x * h(x,n-1)

  Wir verwenden die Induktionshypothese: es gilt $h(x, n-1) = x^{n-1}$

  Somit $h(x,n) = x * h(x,n-1) = x * x^{n-1} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt
- **Induktionsschluss**: somit folgt für alle $n \geq 0$, dass $h(x,n) = x^n$ gilt.

2.3 Rekursive Berechnung der Potenzmenge

Beispiel:
Menge: $M = \{a, b, c\}$
Potenzmenge: $\rho(M) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.1 Rekursiver Lösungsansatz

a. in welchen einfach Fällen kann die Lösung direkt angegeben werden?
   der einfachste Fall ist die leere Menge $M = \emptyset$
   die leere Menge hat nur sich als Teilmenge $\rho(\emptyset) = \{\emptyset\}$
b. Wie können in nicht einfachen Fällen die Teilmengen bestimmt werden?
   Sei Menge $M = \{a_1, \ldots, a_{n-1}, a_n\}$ nicht leer $(n \geq 1)$

1. Wir wählen ein Element der Menge, z.B. a_n
2. Es gibt nun zwei Arten von Teilmengen:

   T^+ Teilmengen, die das Element a_n enthalten

   T^- Teilmengen, die das Element a_n nicht enthalten

   Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^-, d.h. $\rho(M) = T^+ \cup T^-$
   Die Menge T^+ kann nun rekursiv berechnet werden, indem wir a_n aus M entfernen und die Potenzmenge von M berechnen.
   Die Menge T^- ist die Potenzmenge von M ohne a_n.
   Die Potenzmenge von M ist also die Vereinigung von T^+ und T^-.

Beispiel: Wenn $M = \{a, b, c\}$
   Wähle z.B. c als Element:
   T^-: alle Teilmengen ohne c, also alle Teilmengen von $\{a, b\}$
      $T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
   T^+: alle Teilmengen mit c, Nimm zu jeder Teilmenge von T^- und füge c hinzu
      $T^+ = \{\{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$
   Insgesamt: $\rho(\{M\}) = T^+ \cup T^- = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{c\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$

2.3.2 Algorithmischer Ansatz

Falls M leer $(M = \emptyset)$
   leere Menge ist die einzige Teilmenge
Falls M nicht leer Wähle ein Element a_n aus M
   Berechne Sammlung T^- aller Teilmengen von M ohne a_n (rekursiv)
   Berechne Sammlung T^+ aller Teilmengen, die a_n enthalten:
   Nimm dazu jede Menge aus T^- und bilde eine neue Menge, indem a_n hinzugefügt wird
   Die Menge aller Teilmengen ist die Vereinigung von T^+ und T^-

---

```
private static <E> Set<Set<E>> allSubsets(E[] arr,int maxIndex) {
Set<Set<E>> resultSet = new HashSet<Set<E>>();
if (maxIndex >= 0) {
  // Menge ist nicht leer, waehle letzes Element im gegebenen Bereich
E selected = arr[maxIndex];
  // Bilde rekursive alle Teilmengen ohne selected
Set<Set<E>> resultSet1 = allSubsets(arr, maxIndex - 1);
  // nimm jede dieser Mengen zum Ergebnis hinzu
resultSet.addAll(resultSet1);
  // bilde alle Teilmengen, die selected enthalten
for (Set<E> set1 : resultSet1) {
  // Erzeuge Kopie der Menge aus resultSet1 und nimm gewaehltes Element dazu
```

```
Set<E> set2 = new HashSet<E>(set1);
set2.add(selected);
  // fuege die ergaenzte Kopie zum Ergebnis hinzu
resultSet.add(set2);
}
} else {
  // Menge ist leer. Leere Menge hat nur leere Menge als einzige Teilmenge
Set<E> emptySet = new HashSet<>();
resultSet.add(emptySet);
}
return resultSet;
}
```

2.4 Algorithmenprinzip "Backtracking"

2.4.1 Grundidee "Trial and Error"

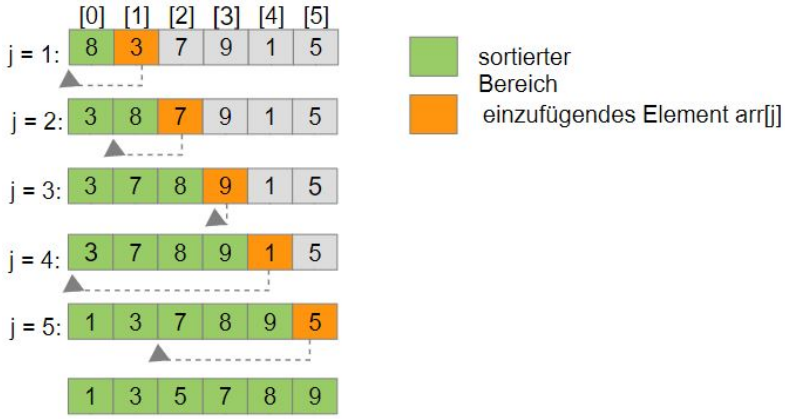
1. Versuche eine Lösung zu finden
2. Wenn die Lösung nicht passt, versuche eine andere Lösung
3. Wenn keine Lösung passt, gehe zurück und versuche eine andere Lösung

3 Analyse von Algorithmen

3.1 Korrektheit

3.1.1 Insertionsort

- Am Anfang des Arrays wird ein sortierter Bereich aufgebaut, in den nach und nach die folgenden Elemente eingefügt werden (deshalb "Insertionsort").
- Am Anfang besteh der sortierte Bereich nur aus dem ersten Element $arr[0]$.
- In jedem Schritt wird ein Element $arr[i]$ aus dem unsortierten Bereich in den sortierten Bereich eingefügt.
- Wenn alle Elemente eingefügt wurden, ist das Array sortiert.



Definition:
Enthält ein Array arr am Anfang die n Elemente $arr[0], arr[1], \ldots, arr[n-1]$, so ist das Array sortiert, wenn gilt:

- **Permutationen:** Die Elemente von arr sind eine Permutattion (Umordnung) der ursprünglichen Elemente von $[0, n-1]$.
- **Monotonie:** Die Elemente von arr sind monoton steigend/fallend sortiert.

Mit **sortiert** ist sofern nicht anders angegeben immer **aufsteigend sortiert** gemeint.

3.2 Komplexität von Algorithmen

Komplexität bezeichnet den Ressourcenverbrauch von Algorithmen. Ressourcen sind dabei die Ausführungszeit und der Speicherbedarf.
Der Ressourcenbedarf hängt von mehreren Faktoren ab:

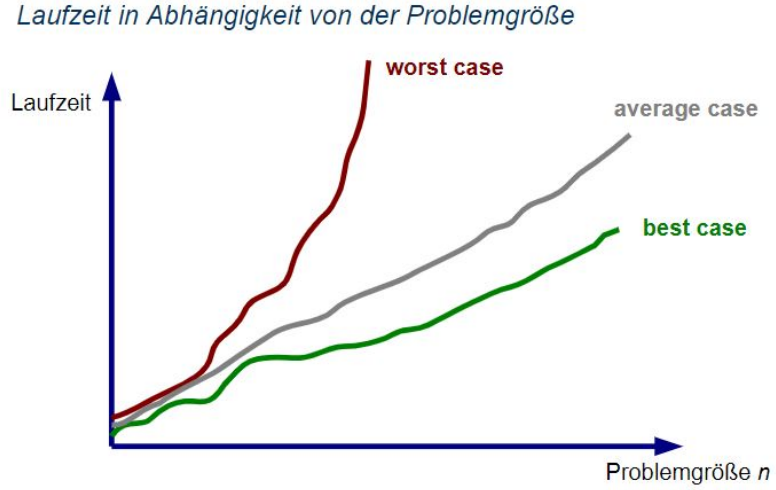
- Umfang der Daten (Größe des Problems)
- Zusammensetzung der Daten (aktuelle Sortierung der Daten)
- Ausführungsgeschwindigkeit und Speicherbedarf bei der Ausführung

3.2.1 Vorgehen bei der Laufzeitanalyse

- Bestimme für jede Anweisung A_j des Programms die Häufigkeit k_j der Ausführung.
- Die Gesamtkosten bei Problemgröße n können dann so zusammengezählt werden: $T(n) = \sum_{A_j}^{n} k_j \cdot c_j$
  wobei k_j die Häufigkeit für die Ausführung von Anweisung A_j ist
  und c_j die Einzel-Ausführungszeit für A_j ist.

3.2.2 Abhängigkeit von der Datenzusammensetzung

Best Case: basiert auf Daten, die im Algorithmus eine minimale Anzahl von Schritten erfordern.
Worst Case: basiert auf Daten, die im Algorithmus eine maximale Anzahl von Schritten erfordern.
Average Case: basiert auf Daten, die im Algorithmus eine durchschnittliche Anzahl von Schritten erfordern.



3.2.3 Häufig verwendete Größenordnungen

- $O(1)$: Konstante Laufzeit - elementare Operationen
- $O(\log n)$: Logarithmische Laufzeit - binäre Suche
- $O(n)$: Lineare Laufzeit - lineare Suche
- $O(n \log n)$: Logarithmische Laufzeit - schnelle Sortieralgorithmen
- $O(n^2)$: Quadratische Laufzeit - einfache Sortieralgorithmen
- $O(n^3)$: Kubische Laufzeit
- $O(c^n)$: Exponentielle Laufzeit
- $O(n!)$: Permutationen berechnen

3.3 Komplexitätsanalyse wohlstrukturierter Algorithmen

Unter wohlstrukturierten Algorithmen versteht man solche, die nur mit Hilfe von Sequenz (nacheinander ausführen), Alternative (Fallunterscheidung) und Iteration (Schleife) definiert sind.

3.3.1 Lineare Suche

Bei der linearen Suche wird ein Array von links nach rechts durchsucht, bis das gesuchte Element gefunden wurde.

```
public String sucheNummer(String suchname) {
  for (int i = 0; i < anzahl; i++) {
    if (liste[i].name.equals(suchname))
    return liste[i].nummer;} // gefunden
return null;} // nicht gefunden
```

3.3.2 Binäre Suche

Bei sortierten Daten kann der Bereich, in dem der gesuchte Schlüssel liegen kann, nach und nach immer wieder halbiert werden. Dieses Verfahren wird **binäre Suche** genannt.

```
private String searchBinRek(String sname, int from, int to){
  if (to < from) {
    return null; //leerer Suchbereich, nichts gefunden
  } else {
  // Mitte des Suchbereichs berechnen
  int middle = (from + to) / 2;
  // Element in der Mitte vergleichen
  int res = suchname.compareTo(liste[middle].name);
    if (res < 0) {
    // in unterer Haelfte weitersuchen
    return searchBinRek(sname, from, middle-1);
    } else if (res > 0) {
    // in oberer Haelfte weitersuchen
    return searchBinRek(sname, middle+1, to);
    } else // (res == 0) {
    // Schluessel gefunden
  return liste[middle].number;
}}}
```

4 Sortierverfahren

4.1 Vergleichsbasierte Sortierverfahren

Interne und Externe Sortierverfahren

- **Interne Sortierverfahren:** alle Datensätze passen in den Hauptspeicher.
- **Externe Sortierverfahren:** nur ein Teil der Datensätze passt in den Hauptspeicher.

Eine Ordnungsrelation heißt **total** wenn sie

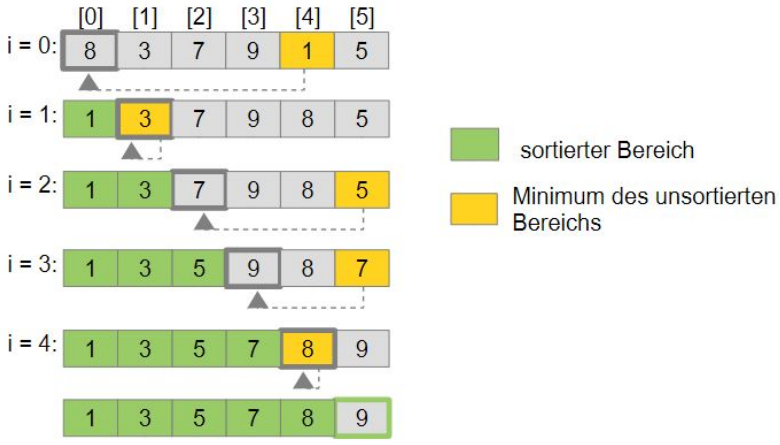
- die Eigenschaften einer Halbordnung erfüllt (reflexiv, transitiv, antisymmetrisch)
  - reflexiv: $a \leq a$
  - transitiv: $a \leq b$ und $b \leq c \Rightarrow a \leq c$
  - antisymmetrisch: $a \leq b$ und $b \leq a \Rightarrow a = b$
- und alle Werte miteinander vergleichbar sind.

Vergleiche über Interface Comparable

```
public interface Comparable {
  int compareTo(Object o) throws ClassCastException;
}
```

wenn $x < y$ dann $x.compareTo(y) < 0$
wenn $x = y$ dann $x.compareTo(y) = 0$
wenn $x > y$ dann $x.compareTo(y) > 0$

4.1.1 Laufzeitkomplexität von Selection-Sort



Worst Case = Best Case = Average Case: $O(n^2)$

4.1.2 Laufzeitkomplexität von Bubble-Sort

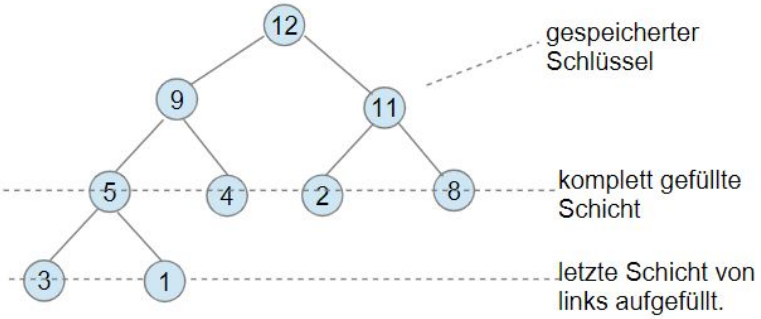
Best Case: $O(n)$ - Daten sind schon aufsteigend sortiert
Worst Case: $O(n^2)$ - Daten sind absteigend sortiert
Average Case: $O(n^2)$

4.2 Heapsort

4.2.1 Binäre Maximum-Heaps

Ein **Maximum-Heap** ist ein binärer Baum, der folgende Eigenschaften erfüllt:

- Der Baum ist fast vollständig - alle Ebenen bis auf die letzte sind vollständig.
- Für jeden Knoten gilt, dass der Schlüssel des Knotens größer oder gleich dem Schlüssel der Kinder ist.



Speicherung eines fast vollständigen Baumes in einem Array:
Die Wurzel wird an Index 0 abgelegt
Hat ein Knoten den Index i, so sind die Kinder an den Indizes $2i + 1$ und $2i + 2$ abgelegt.

4.2.2 Soriteren mittels Heap

Grundprinzip von Heapsort:
1. Heap aufbauen: Array wird in einen Maximum-Heap umgewandelt.
2. Heap abbauen, sortierten Bereich aufbauen:

- Vertausche Wurzel a[0] und a[k], Maximum ist nun am Ende des Arrays.
- Heap-Eigenschaft wiederherstellen: Wurzel a[0] nach unten schieben, bis Heap-Eigenschaft wiederhergestellt ist.

4.2.3 Laufzeitkomplexität von Heapsort

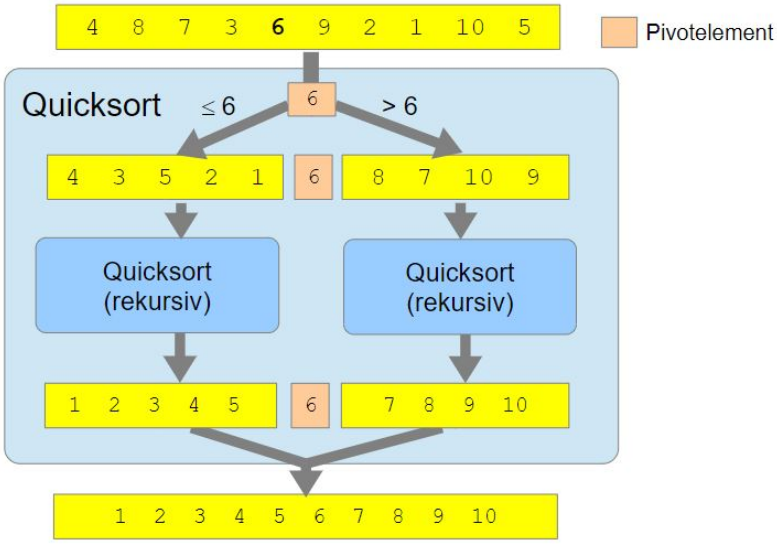
"versenken" (sink): $O(\log n)$
"erstelleMaxHeap" (createMaxHeap): $O(n \log n)$
Heapsort: $O(n \log n)$

4.3 Rekursives Sortieren nach "Teile und Herrsche"

1. **Divide**: Besteht die Datenmenge aus mehr als einem Element, so wird sie in zwei Teilmengen zerlegt.
2. **Conquer**: Die Teilmengen werden rekursiv sortiert.
3. **Combine**: Die Teilmengen werden zusammengeführt.

divide	*combine*	*Sortierverfahren*
Aufteilen in Werte, die größer bzw. kleiner als ein gewähltes Vergleichselement (sog. Pivotelement) sind	Sortierte Teilfolgen aneinander anhängen (bzw. stehen schon nacheinander im Feld), mit Pivotelement dazwischen.	⇒ Quicksort
Aufteilen der zu sortierenden Werte in zwei gleich große Teilmengen (z.B. Arraybereich in der Mitte teilen).	Verschmelzen der sortierten Teilfolgen zu einer sortierten Gesamtfolge durch merge-Operation	⇒ Mergesort

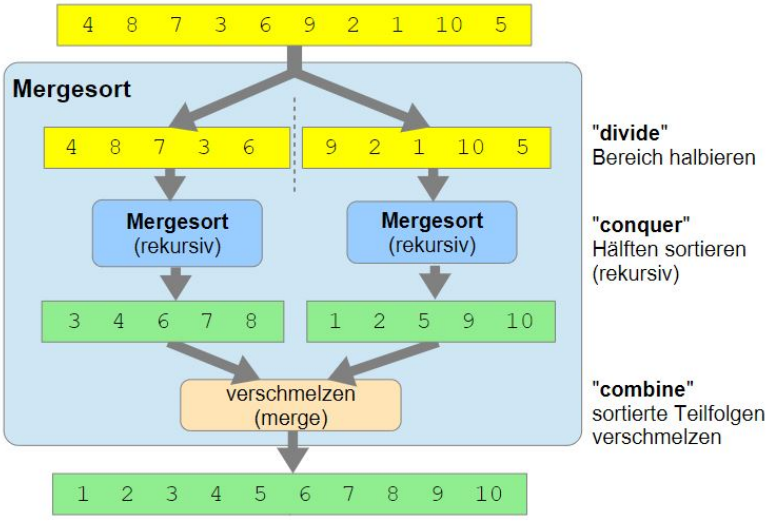
4.4 Quicksort



4.4.1 Laufzeitkomplexität von Quicksort

Worst Case: $O(n^2)$
Best Case: $O(n \log n)$
Average Case: $O(n \log n)$

4.5 Mergesort




```
void mergesort(double[] a, int von, int bis) {
  if (bis - von > 0) {
  //Mehr als ein Element zu sortieren
  //Daten in zwei Haelften aufteilen
  int mitte = (von + bis) / 2;
  //linke und rechte Haelfte sortieren
  mergesort(a, von, mitte);
  mergesort(a, mitte+1, bis);
  // Sortierte Teilfolgen a[von..mitte] und
  // a[mitte+1..bis] miteinender verschmelzen
  merge(a, von, mitte, bis);
  }
```

```
}
```

4.5.1 merge - Verschmelzen sortierter Teilfolgen

```
void merge(double[] a, int links, int mitte, int rechts) {
  //Kopie der linken Haelfte a[links..mitte] erzeugen
  double[] tmpLinks = new double[mitte - links + 1];
  for (int i = 0; i < tmpLinks.length; i++) {
    tmpLinks[i] = a[links + i];
  }
  // Inhalt von tmpLinks (linker Teil) und a[mitte+1 .. rechts] zu
  // sortierter Gesamtfolge im Ergebnisbereich a[links..rechts] verschmelzen
  int indexLinks = 0;
  int indexRechts = mitte+1;
  int indexErgebnis = links;
  while (indexLinks < tmpLinks.length && indexRechts <= rechts) {
    // linke und rechte Teilfolge enthalten noch Elemente
    // nimm kleineren Wert von beiden Teilfolgen als naechsten Wert
    if (tmpLinks[indexLinks] <= a[indexRechts]) {
      a[indexErgebnis] = tmpLinks[indexLinks];
      indexLinks++;
    } else {
      a[indexErgebnis] = a[indexRechts];
      indexRechts++;
      }
    indexErgebnis++;
  }
  // falls nur noch linke Teilfolge Werte enthaelt (rechte Teilfolge aufgebraucht),
  // uebertrage sie in das Ergebnis
  while (indexLinks < tmpLinks.length) {
    a[indexErgebnis] = tmpLinks[indexLinks];
    indexErgebnis++;
    indexLinks++;
  }
  // falls linke Teilfolge abgearbeitet ist und nur noch rechte Teilfolge Werte
  // enthaelt, stehen diese schon richtig im Ergebnisfeld a und muessen nicht
  // mehr behandelt werden
}
```

4.5.2 Laufzeitkomplexität von Mergesort

Worst Case = Best Case = Average Case: $O(n \log n)$

4.6 Eigenschaften der Sortierverfahren

Stabile Sortierverfahren: Ein Sortierverfahren heißt stabil, wenn die relative Reihenfolge von Elementen, deren Schlüssel gleich sind, während des Sortiervorgangs beibehalten wird wird.
Bubble-Sort, Insertion-Sort, Mergesort sind stabile Sortierverfahren.
Quicksort, Heapsort sind instabile Sortierverfahren.
Selectionsort ist ein instabiles Sortierverfahren, das aber stabil gemacht werden kann.
(für Arrays instabil -¿ für Listen stabil)

4.6.1 Zusammenfassung der Sortierverfahren

Verfahren	*best case*	*average case*	*worst case*	*stabil*	*in-place-Verfahren*
Bubblesort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	ja	ja
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	nein[1)]	ja
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	ja	ja
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	nein	ja
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	ja	nein
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	nein	ja

4.7 Nicht-vergleichsbasierte Sortierverfahren

Jedes vergleichsbasierte Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln im schlechtesten Fall (und auch im mittleren Fall) $O(n \log n)$ Schlüsselvergleiche.

4.7.1 Eigenschaften eines Entscheidungsbaums

- Ein Binärbaum repräsentiert die beim Suchen durchgeführten Vergleiche.
- Wurzelknoten repräsentiert den ersten Vergleich.
- Verzweigung nach links repräsentiert den Fall, dass der Vergleich wahr ist.
- Der Worst-Case ist der Pfad mit der größten Tiefe.

4.7.2 Bucket-Sort

Idee: Die Elemente werden in Buckets sortiert, die jeweils einen Bereich von Schlüsseln repräsentieren. Die Buckets werden dann einzeln sortiert und anschließend zusammengeführt.

gegeben:

- Schlüsselbereich [0, 100[
- n = 12 zu sortierende Werte: 78, 17, 39, 26, 72, 94, 21, 12, 23, 68, 77, 42
- vorgegebene Konstante c = 3

Sortieren mit Bucketsort:

- Es sind $m = n/c = 12/3 = 4$ Fächer zu bilden.
- Berechnung des Fachindex für Schlüssel k nach der Formel $F(k) = \lfloor k/25 \rfloor$
- Ablauf:

(1) Werte durchgehen und auf Fächer (buckets) verteilen:

Fachindex	Schlüsselbereich für Fach	Werteliste
0	[0, 25[	17, 21, 12, 23
1	[25, 50[	39, 26, 42
2	[50, 75[	72, 68,
3	[75, 100[	78,94, 77

(2) Fächer sortieren mit üblichem Sortierverfahren (z.B. Insertionsort)

Fachindex	Schlüsselbereich für Fach	Werteliste
0	[0, 25[	12, 17, 21, 23
1	[25, 50[	26, 39, 42
2	[50, 75[	68, 72
3	[75, 100[	77, 78, 94

(3) Fächer aufsteigend durchgehen und Werte einsammeln.
Ergebnis:

12, 17, 21, 23, 26, 39, 42, 68, 72, 77, 78, 94

Laufzeitkomplexität:
Best Case = Average Case: $O(n)$
Worst Case $O(n^2)$

4.7.3 Radix-Sort

Wiederhole für jede Stelle p der insgesamt d Stellen, von der niederwertigsten Stelle rechts bis zur höchstwertigen Stelle links jeweils folgende Partitioniserungs- und Sammelphase
(1) Partitionierungsphase: Gehe die Wertefolge durch und ordne sie entsprechend dem Zeichen an der Stelle p in das zugehörige Fach ein.
wichtig: neue Elemente werden hinten an die Fachliste angehängt!
(2) Sammelphase: Die Einträge aus den Fächern aufsammeln und wieder in eine lineare Folge bringen
- bei Fach mit niedrigstem Wert beginnen
- die im Fach vorgegebene Reihenfolge muss beibehalten werden
Diese Folge wird dann beim nächsten Durchgang für die Partitionierung verwendet.
Nach d Durchgängen durch alle Stellen sind die Daten richtig sortiert.
Laufzeitkomplexität:
Best Case = Worst Case = Average Case: $O(d \cdot n) = O(n)$

5 Einfache Datenstrukturen

5.1 Abstrakte und konkrete Datenstrukturen

Abstrakte Datentypen (ADT):
Ein ADT definiert eine Datenstruktur unabhängig von ihrer konkreten Implementierung.
Konkrete Datenstrukturen:
Ein konkreter Datentyp ist die Implementierung eines ADT.

5.2 Abstrakter Datentyp Stack

Stack: Eine Sammlung von Elementen, die nach dem LIFO-Prinzip (Last In First Out) verwaltet wird.

```
public interface Stack {
  void push(Object elem); // Element oben auf dem Stack ablegen
  Object pop() throws EmptyStackError;
  // oberstes Element des Stacks entnehmen und zurueckgeben
  boolean isEmpty(); // prueft, ob Stack leer ist
}
```

5.3 Konkreter Datentyp ArrayStack

Bei Stacks kann man noch zwei Varianten unterscheiden:
ob sie eine begrenzte Kapazität haben oder sie unbeschränkt wachsen können.
Laufzeitkomplexität für ArrayStack mit begrenzter Kapazität: $O(1)$

5.4 Amortisierte Laufzeitanalyse

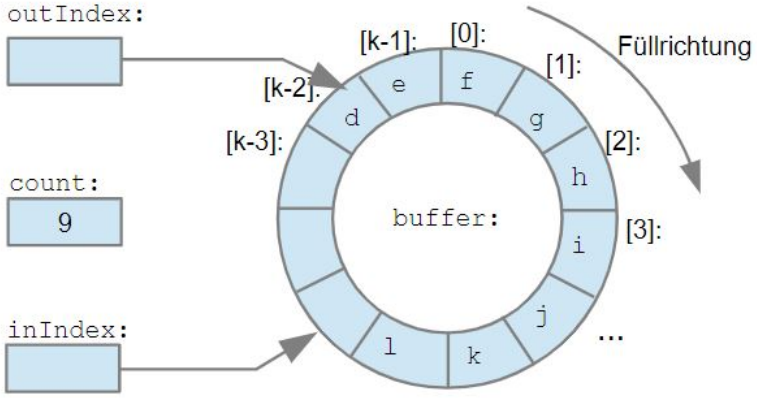
Die Laufzeit eines Algorithmus wird über mehrere Operationen hinweg betrachtet.

5.5 Abstrakter Datentyp Warteschlange

Eine Queue ist eine Sammlung von Elementen, die nach dem FIFO-Prinzip (First In First Out) verwaltet wird.

```
public interface Queue {
  void enqueue(Object elem); // fuege Element vorne ein
  Object dequeue() throws EmptyQueueError;
  // entnimmt Element hinten
  boolean isEmpty(); // pruefen, ob Wartschlange leer ist
}
```

5.6 Konkreter Datentyp Ringpuffer



Laufzeitkomplexität: $O(1)$S

5.7 Abstrakter Datentyp Prioritätswarteschlange

Eine Prioritätswarteschlange ist eine Datenstruktur zur Verwaltung eine Sammlung von Einträgen, wobei die Einträge jeweils mit einer Priorität versehen sind, die bestimmt, in welcher Reihenfolge die Elemente entnommen werden.
Grundoperationen:
insert(x) - Einfügen eines ELements (Schlüssel in x enthalten)
extractMin() - Rückgabe & Entfernen des Elements mit kleinstem Schlüssel –¿ höchste Priorität

5.8 Konkreter Datentyp verkettete Liste

Die Datenstruktur, die eine Folge von Einträgen speichert, wird durch einzelne Knotenobjekte realisiert, die jeweils über eine Referenz auf einen Nachfolger verweisen. Durch Anhängen, Entfernen oder Umhängen einzelner Knotenobjekte kann die Wertefolge dynamisch in Umfang und Reihenfolge verändert werden.
Einfach verkette Liste mit first- und last-Referenz
Doppelt verkettete Liste mit first- und last-Referenz

5.9 Abstrakter Datentyp Double-Ended-Queue (Deque)

Eine Deque ist eine Warteschlange, bei der an beiden Enden Elemente eingefügt und entfernt werden können.

5.10 Zusammenfassung: Anwendung verketter Listen

- Stacks
  push(v), pop() - $O(1)$
- Queues
  enqueue(v), dequeue() - $O(1)$
- Double-Ended-Queues
  insertFirst(v), insertLast(v), removeFirst(), removeLast() - $O(1)$

6 Suchbäume

6.1 Abstrakte Datentypen Map und Set

6.1.1 Abstrakter Datentyp Map

Ein Map ist eine Sammlung von Schlüssel-Wert-Paaren, die nach den Schlüsseln verwaltet wird.
Grundoperationen:
put(k,v) - Einfügen eines Schlüssel-Wert-Paares - v wird unter dem Schlüssel k abgelegt
get(k) - Rückgabe des Wertes, der unter dem Schlüssel k abgelegt ist
delete(k) - Entfernen des Schlüssel-Wert-Paares mit Schlüssel k

6.1.2 Abstrakter Datentyp Set

Ein Set ist eine Sammlung von Elementen, die nach den Elementen verwaltet wird.
- Ein Element kann höchstens einmal im Set vorkommen
Grundoperationen:
add(x) - Einfügen eines Elements
remove(x) - Entfernen eines Elements
contains(x) - Prüfen, ob ein Element im Set enthalten ist
isEmpty() - Prüfen, ob das Set leer ist
size - Anzahl der Elemente im Set

6.2 Binäre Bäume

6.2.1 Implementierung eines binären Baumes

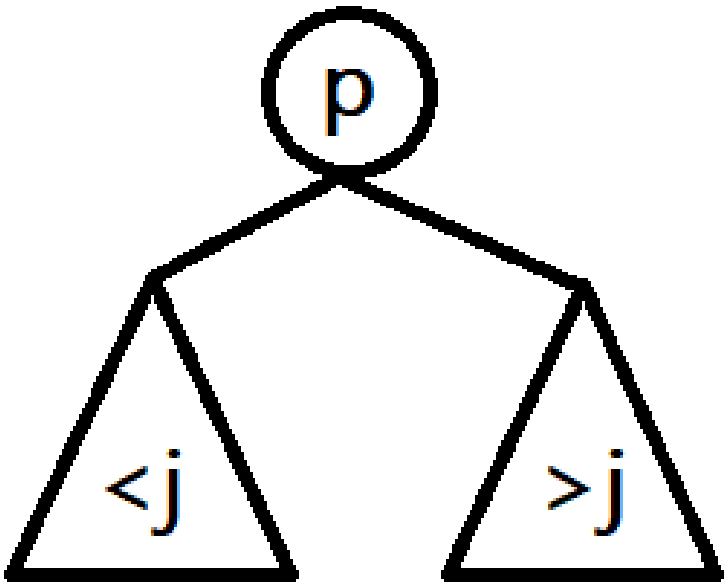
```
public class TreeNode {
  Object value; // Wert des Knotens
  TreeNode left, right; // linke und rechte Teilbaeume
  public TreeNode(Object value, TreeNode left, TreeNode right) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
  public TreeNode(Object value) { // erzeugt Blattknoten
    this(value, null, null);
  }
}
```

Traversierung eines binären Baumes:
Inorder: links, Wurzel, rechts - sortiert
Preorder: Wurzel, links, rechts - für Ausdrücke
Postorder: links, rechts, Wurzel - für Ausdrücke

6.3 Binäre Suchbäume

Ein Suchbaum ist ein binärer Baum, bei dem für jeden Knoten p des Baums gilt:
- Alle Schlüssel im linken Teilbaum von p sind kleiner als der Schlüssel von p

- Alle Schlüssel im rechten Teilbaum von p sind größer als der Schlüssel von p



6.3.1 Suchbaum-Operationen

Rekursive Suche:

```
Suche nach Knoten mit Key k
- falls Baum leer: Suche fehlgeschlagen
- falls Baum nicht leer, Wurzel hat Key w:
  - falls w = k: Suche erfolgreich
  - falls k < w: Suche im linken Teilbaum
  - falls k > w: Suche im rechten Teilbaum
```

Einfügen eines Knotens:

```
Einfuegen eines Knotens mit Key k
- falls Baum leer: neuer Knoten wird Wurzel
- falls Baum nicht leer, Wurzel hat Key w:
  - falls k < w: Einfuegen im linken Teilbaum
  - falls k > w: Einfuegen im rechten Teilbaum
```

Entfernen eines Knotens:

```
Entfernen eines Knotens mit Key k
- falls Baum leer: Entfernen fehlgeschlagen
- falls Baum nicht leer, Wurzel hat Key w:
  - falls w = k: Entfernen erfolgreich
  - falls k < w: Entfernen im linken Teilbaum
  - falls k > w: Entfernen im rechten Teilbaum
```

6.3.2 Laufzeitkomplexität von binären Suchbäumen

- Suche: $O(\log n)$ - Worst Case: $O(n)$
- Einfügen: $O(\log n)$ - Worst Case: $O(n)$
- Entfernen: $O(\log n)$ - Worst Case: $O(n)$

6.4 Grundprizip balancierter Suchbäume

Bäume werden durch Rotationen so verändert, dass die Tiefe der Teilbäume möglichst gleich bleibt.

6.5 AVL-Bäume

6.5.1 Balancefaktor und AVL-Ausgleich

Balancefaktor:

$$B_x = height(left(x)) - height(right(x))$$

Ein Blatt hat Balancefaktor 0, da die Höhe des linken und rechten Teilbaums gleich ist.
AVL-Ausgleich:
Ein AVL-Baum ist ein binärer Suchbaum, bei dem für jeden Knoten p gilt:
- Der Balancefaktor von p ist -1, 0 oder 1
Höhe eines AVL-Baums:

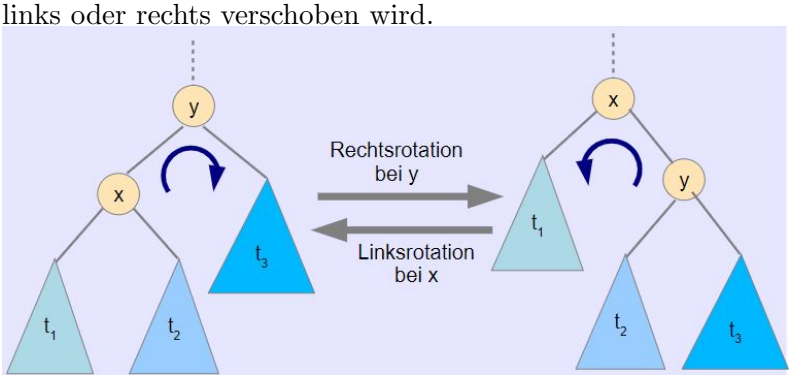
$$h = O(\log n)$$

6.5.2 Suche in AVL-Bäumen

Die Laufzeitkomplexität von AVL-Bäumen ist im Worst und Average Case $O(\log n)$.

6.5.3 Rotationen in Suchbäumen

Eine Links- oder Rechtsrotation ist eine Rotation, bei der ein Knoten p um eine Ebene nach links oder rechts verschoben wird.



6.5.4 Einfügen in AVL-Bäumen

```
(1) Suche wie beim Suchbaum die Position k (als Blatt)
(2) Einfuegen des neuen Knotens k
(3) Falls Balancefaktor von k ungleich 0:
  - Falls Balancefaktor von k ungleich Balancefaktor von p:
    - Falls Balancefaktor von k = -1 und Balancefaktor von p = -1:
      - Rechtsrotation um p
    - Falls Balancefaktor von k = 1 und Balancefaktor von p = 1:
      - Linksrotation um p
  - Falls Balancefaktor von k gleich Balancefaktor von p:
    - Falls Balancefaktor von k = -1 und Balancefaktor von p = -1:
      - Rechtsrotation um p
      - Linksrotation um q
    - Falls Balancefaktor von k = 1 und Balancefaktor von p = 1:
      - Linksrotation um p
      - Rechtsrotation um q
```

6.5.5 Entfernen in AVL-Bäumen

```
(1) Suche den zu entfernenden Knoten k und merke den Pfad von der Wurzel zu k.
(2) Falls Knoten k Blatt ist oder nur ein Kind hat:
       Entferne k wie in einem Suchbaum
    Falls Knoten k zwei Kinder hat:
       Suche im linken Baum von k den groessten Knoten l und ersetze k durch l
(3) Verfolge den Pfad zur Wurzel und aktualisiere die Balancefaktoren und rotiere bei
    Bedarf
(4) Wenn AVL-Ausgleich verletzt ist --> Einfach- bzw Doppelrotation
```

6.5.6 Laufzeitkomplexität für AVL-Bäume

Worst & Average Case:

- Suche: $O(\log n)$
- Einfügen: $O(\log n)$
- Entfernen: $O(\log n)$

6.6 B-Bäume

Es handelt sich dabei um Mehrwegbäume, bei denen ein einzelner Knoten eine (sortierte) Folge von Schlüsseln speichert und es auch entsprechendmehr als zwei Kinder geben kann.

6.6.1 Definition

Für einen B-Baum der Ordnung m ($m \geq 2$) gilt:
Struktur des Baums:

(1) Alle Blätter haben die gleiche Tiefe - Abstand zur Wurzel

(2) Enthält ein innerer Knoten j Schlüssel, dann hat er j+1 Kinder

(3) Jeder Knoten enthält maximal $2m - 1$ Schlüssel

(4) Jeder Knoten enthält mindestens $m - 1$ Schlüssel

(5) Die Wurzel enthält mindestens einen Schlüssel

Geordnete Speicherung:
Enthält ein Knoten die Schlüssel $k_1, k_2, \ldots, k_n$, dann gilt:

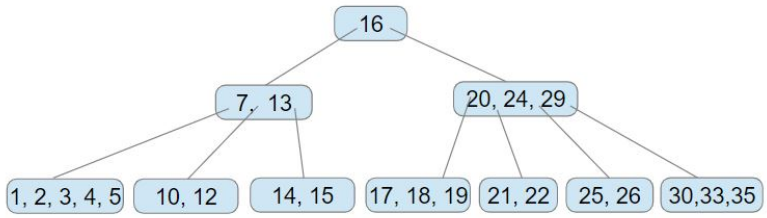
(6) Die Schlüssel sind aufsteigend geordnet

(7) Für alle Werte x im ersten Unterbaum des Knotens gilt: $x < k_1$

(8) Sind k_{j-1} und k_j zwei aufeinanderfolgende Schlüssel, dann gilt: $k_{j-1} < x < k_j$

(9) Für alle Werte x im letzten Unterbaum des Knotens gilt: $x > k_j$

Folgender Baum ist ein B-Baum der Ordnung $m = 3$:



- minimal m-1 = 2 Schlüssel (außer Wurzel), d.h. minimal m = 3 Kinder
- maximal 2m-1 = 5 Schlüssel, d.h. maximal 2m = 6 Kinder

6.6.2 Höhe von B-Bäumen

$h \leq [log_m(\frac{n+1}{2})]$

6.6.3 2-3-4 Bäume

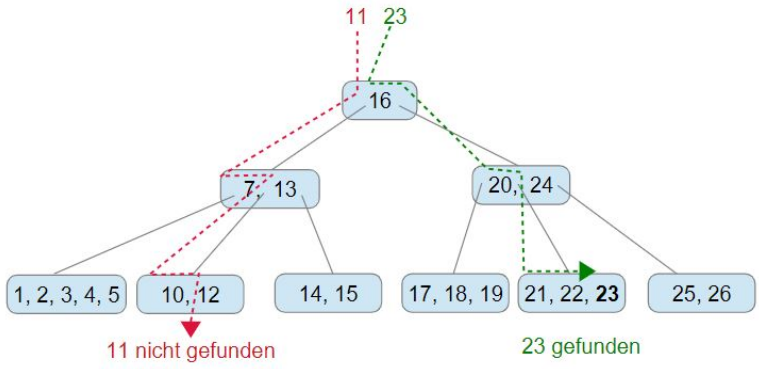
B-Bäume der Ordnung $m = 2$ werden auch 2-3-4 Bäume genannt.

- Jeder Knoten kann zwei, drei oder vier Nachfolger haben
- Jeder Knoten kann einen, zwei oder drei Schlüssel enthalten

6.6.4 Suche in B-Bäumen

```
- Man beginnt mit dem Wurzelknoten
- In einem Knoten kann mit bin. Suche nach dem Wert gesucht werden
- Wenn der Wert gefunden wurde, ist die Suche beendet
- Wenn der Wert nicht gefunden wurde, wird der entsprechende Unterbaum
  durchsucht
- Wenn der Wert nicht gefunden wurde, ist die Suche beendet
```

Suche nach den Schlüsseln 11 und 23:



6.6.5 Einfügen in B-Bäumen

```
- Ausgehend von der Wurzel das passende Blatt fuer Key k suchen
- Neuen Key k an der richtigen Stelle in den Blattknoten einfuegen
  - Falls Blatt schon voll ist, muss der Knoten aufgeteilt werden
```

Beweis mittels Berechnungsinduktion. Eigenschaft, die nachgewiesen werden soll:

$P((x,n),y) :\Leftrightarrow x^n = y$

d.h. wir wollen zeigen, dass $h(x,n) = x^n$ gilt.

- **Induktionsbasis** (Argumente führen zu keinem rekursiven Aufruf)
  Fall n = 0:
  Ergebnis: $h(x, n) = 1 = x^0$, d.h. Eigenschaft erfüllt
- **Induktionsschritte** (Argumente führen zu rekursiven Aufrufen):
  Wir können als Induktionshypothese für die rekursiven Aufrufe verwenden, dass $h(z_i,k_i) = z_i^{ki}$ gilt.

  Fall n > 0, n gerade:
  Ergebnis laut Programmcode: $h(x,n) = y*y$, wobei $y = h(x, n/2)$

  Wir verwenden die Induktionshypothese: es gilt $h(x, n/2) = x^{n/2}$

  somit $h(x,n) = y*y = x^{n/2} * x^{n/2} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt.

  Fall n > 0, n ungerade:
  Ergebnis laut Programmcode: $h(x,n) = x * h(x,n-1)$

  Wir verwenden die Induktionshypothese: es gilt $h(x, n-1) = x^{n-1}$

  Somit $h(x,n) = x * h(x,n-1) = x * x^{n-1} = x^n$,
  d.h. Eigenschaft in diesem Fall auch für *n* erfüllt
- **Induktionsschluss**: somit folgt für alle $n \geq 0$, dass $h(x,n) = x^n$ gilt.

6.6.6 Entfernen in B-Bäumen

Schlüssel liegt im Blatt:
Der Schlüssel wird entfernt und der Knoten wird ggf. zusammengeführt.
Schlüssel liegt nicht im Blatt:
Der Schlüssel wird durch seinen direkten Vorgänger (Maximum davor) oder direkten Nachfolger (Minimum danach) ersetzt und der Knoten wird ggf. zusammengeführt.

6.6.7 Laufzeitkomplexität für B-Bäume

Worst & Average Case:

- Suche: $O(\log n)$
- Einfügen: $O(\log n)$
- Entfernen: $O(\log n)$

6.7 Rot-Schwarz Bäume

Binäre Suchbäume haben den Vorteil, dass sie einfach zu implementieren sind und ein einfaches Suchen und Traversieren ermöglichen.
B-Bäume haben den Vorteil, dass sie immer weitgehend ausbalanciert sind.
Rot-Schwarz Bäume kombinieren die Vorteile der beiden.

6.7.1 Definition

- Jeder Knoten ist rot oder schwarz
- Die Wurzel ist schwarz
- Jeder rote Knoten hat nur schwarze Kinder
- Jeder Nil-Knoten unter einem Blatt ist per Definition schwarz
- Für jeden Knoten k gilt: Jeder Pfad von k zu einem Blatt hat die gleiche Anzahl an schwarzen Knoten

- Nil-Knoten: Knoten ohne Schlüssel - nur als Platzhalter (schwarz)

Höhe von Rot-Schwarz Bäumen: $h \leq 2 \cdot [log_2(n+1)]$

6.7.2 Suche in Rot-Schwarz Bäumen

Die Suche erfolgt wie bei binören Suchbäumen - die Farbe der Knoten spielt dabei keine Rolle.
Worst Case: $O(\log n)$

6.7.3 Einfügen in Rot-Schwarz Bäumen

Beim Einfügen neuer Schlüssel kann zunächst, wie bei binären Suchbäumen üblich, vorgegangen werden und der Wert als neues Blatt an passender Stelle eingehängt werden.
Danach muss die Farbe des Knotens und die Farbe der Kinder geprüft werden.

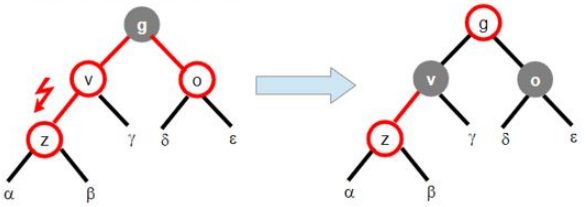
- Wird ein neues Blatt als roter Knoten eingefügt führt das ggf. zu einem Rot-rot-Problem
- Wird ein neues Blatt als schwarzer Knoten eingefügt, zerstört das die Eigenschaft, dass auf allen Pfaden von der Wurzel zu einem Blatt die gleiche Anzahl an schwarzen Knoten liegt
- Probleme mit roten Blättern sind leichter zu lösen als mit schwarzen Blättern

Prinzipielle Vorgehensweise:

```
- neuen Key als Blatt einfuegen - Farbe rot (2 schwarze Nil-Knoten)
- Falls Elternknoten rot --> Korrekturen
- Falls am Ende Wurzel rot --> Farbe schwarz
```

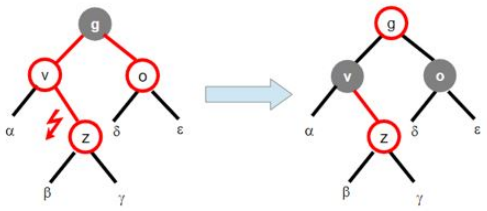
Reorganisation durch Insertion Fixup:

Insert-Fixup(z) - Fall (1) : Problemknoten z ist linkes Kind von v, v ist linkes Kind von g, Onkel o von z ist rot.
Reorganisation: Knoten g, v, o auf Eltern und Großelternebene jeweils umfärben in die andere Farbe.



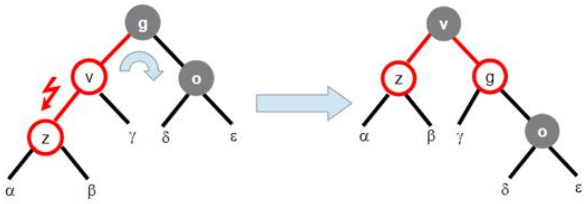
Da der Großvater rot geworden ist, muss danach ggf. Insert-Fixup für den Großvaterknoten g wiederholt werden, falls der Elternknoten von g auch rot ist.

Insert-Fixup(z) - Fall (2): Problemknoten z ist rechtes Kind von v, v ist linkes Kind von g, Onkel o von z ist rot:
Reoganisation: Auch in diesem Fall kann das Problem bei Knoten z durch Umfärben von Vater- Onkel und Großvaterknoten beseitigt werden:



Wie in Fall (1) wird ggf. eine weitere Korrektur per Insert-Fixup für Großvaterknoten g nötig, wenn der Elternknoten von g auch rot ist.

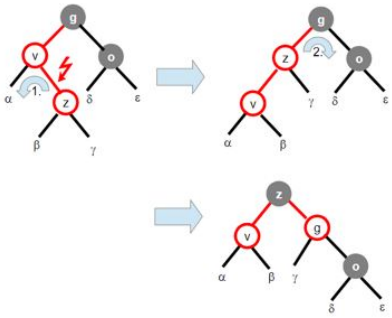
Insert-Fixup(z) - Fall (3): Problemknoten z ist linkes Kind von v, v ist linkes Kind des g, Onkel o ist schwarz.
Reorganisation: Rechtsrotation bei Großvater g und Umfärben der Knoten g und v.



Da nach Rotation und Umfärben die neue Wurzel v des Teilbaums schwarz ist, ist keine weitere Korrektur weiter oben im Baum nötig.

Insert-Fixup(z) - Fall (4): "Problemknoten" z ist rechtes Kind von v, v ist linkes Kind von g, Onkel o von z ist schwarz.

Reorganisation: Es wird ein Doppelrotation ausgeführt. Zunächst Linksrotation bei Vater v, danach weiter wie bei Fall (3), d.h. Rechtsrotation bei g und Umfärben von z und g.



6.7.4 Laufzeitkomplexität für Rot-Schwarz Bäume

Worst & Average Case:
- Einfügen: $O(\log n)$
- Entfernen: $O(\log n)$
- Suche: $O(\log n)$